Computer Science Department

TECHNICAL REPORT

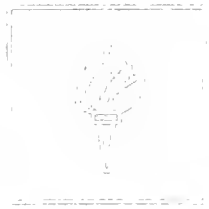
THE FIVE COLOR CONCURRENCY CONTROL PROTOCOL: NON-TWO-PHASE LOCKING IN GENERAL DATABASES

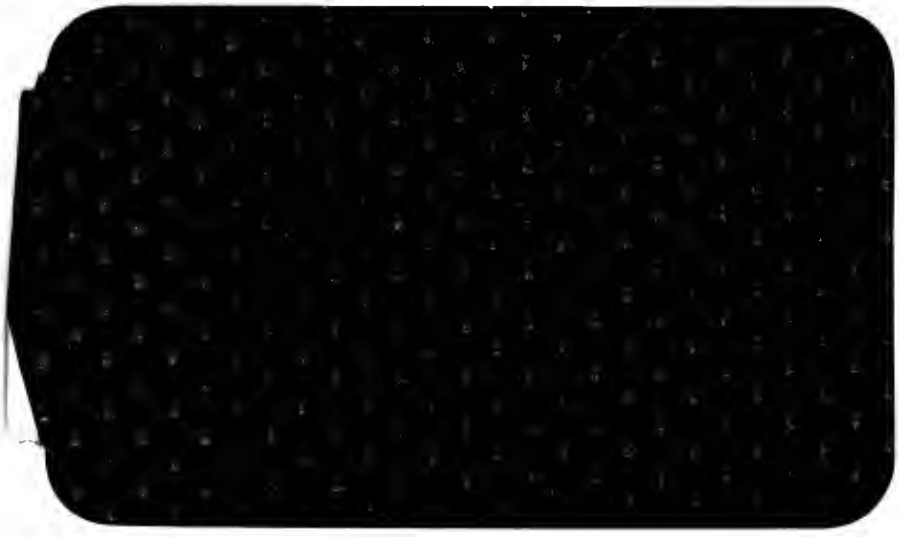
By

Partha Dasgupta and Zvi M. Kedem‡

Technical Report #213
April, 1986

NEW YORK UNIVERSITY





THE FIVE COLOR CONCURRENCY CONTROL PROTOCOL: NON-TWO-PHASE LOCKING IN GENERAL DATABASES

By

Partha Dasgupta and Zvi M. Kedem ‡

Technical Report #213
April, 1986

* Partially supported by NASA under contract number NAG-1-30, by NSF under grant numbers MCS 81-04882, MCS 81-1009*, and DCR-8416422, and by ONR under contract number N00014-85-K0046

‡ Authors' Addresses.

Partha Dasgupta
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

Zvi M. Kedem
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

The *Five Color* Concurrency Control Protocol: Non-Two-Phase Locking in General Database

Partha Dasgupta and Zvi M. Kedem

Abstract

Concurrency control protocols that are based on 2-phase locking are a popular family of locking protocols that preserved serializability in general (unstructured) database systems. This paper presents a concurrency control algorithm (for databases with no inherent structure) that is practical, non-2-phase and allows varieties of serializable logs not possible with any commonly known locking schemes. The protocol achieves high concurrency by anticipating the existence (or absence) of possible conflicts using information about transaction read and write sets. It is well known that serializability is characterized by acyclicity of the conflict graph representation of interleaved executions. The 2-phase locking protocols allow only *forward* growth of the paths in the graph. The *Five Color* protocol allows the conflict graph to grow in any direction (avoiding 2-phase constraints) and prevents cycles in the graph by maintaining transaction access information in the form of data-item markers. The read and write set information can also be used to provide relative immunity from deadlocks. This protocol allows higher concurrency and lower deadlock frequencies than 2-phase locking, according to our simulation studies.

1 Introduction.

This paper presents a concurrency control mechanism that uses five kinds of locks. Unlike the 2-phase locking protocol, the *Five Color* protocol uses early release of locks to enhance concurrency. The early release of locks causes the protocol to be non-two phase in its locking behavior. It has been shown that 2-phase locking is a necessary condition for serializability in general databases. However we show how serializability can be achieved using a non-two phase protocol by addition of a validation phase. The *Five Color* protocol does not assume any inherent structures in the database. We first present a brief introduction to the to the concepts of serializability, the model of a multiuser database, the factors that limit concurrency in 2-phase locking and some related work. Section 2 contains a comprehensive description of the *Five Color* protocol, including an intuitive description of how it functions and why it ensures serializability. Section 3 explains the formal properties of the protocol and derives a proof of correctness. Sections 5 and 6 deal with

deadlocks and livelocks and Section 7 compares the efficiency of this protocol to other well known protocols, using both intuitive reasoning and simulation results.

1.1 Serializability.

A database is viewed as a collection of data items, which can be read or written by concurrent transactions. Interleaving of updates can leave the database in an inconsistent state. A sufficient condition to guarantee correctness of concurrent database access is *serializability* of the actions (reads or writes) performed by the transactions on the data items. That is, the interleaved execution of the transactions should be equivalent to any serial execution of the transactions [BeGo80, RoStLe78]. In this paper, we assume serializability to be the criterion of correctness.

Locking of data items is one of the methods of achieving consistency in the face of concurrent updates. For databases with no inherent structure (e.g. databases not organized as DAG's, trees, etc. the 2-phase locking protocol is the most popular locking protocol. However, 2-phase locking is restrictive with respect to the amount of concurrency it allows.

Informally, a *log* is a sequence of actions issued by various transactions on several data items. The transaction actions may be interleaved with one another. Serializability is a syntactic property of a log. It has been shown that recognizing serializability is an NP-complete problem [BeShWo79, Pa79]. The NP-completeness of the serializability recognition problem implies that we cannot have a scheduler that allows all serializable logs and disallows non-serializable ones, and works in polynomial time (unless P=NP). However certain subclasses of serializable logs are efficiently recognizable in polynomial time. Efficient algorithms can be built that control the actions of transactions, to ensure that the logs produced by a set of transactions fall into one of these easily recognizable classes of serializability. The 2-phase locking protocol is one such algorithm which produces a class of polynomially recognizable serializable logs, namely the 2-phase locked logs.

1.2 The Model.

A *database* **D** is a set of distinct items $\{x_1, x_2, \ldots, x_m\}$. A *transaction system* **T** is a set of transactions $\{T_1, T_2, \ldots, T_n\}$ that operate on the database. The *readset* (*writeset*) of a transaction T_i is the set of all items T_i reads (writes).

A transaction that intends to read (or write) a data item x, issues a read (or write) request to the *transaction manager*. The transaction manager is responsible for determining whether or not granting of the request may cause a violation of the correctness criterion (generally serializability). The transaction manager then takes appropriate action by granting, rejecting or delaying the request.

A *trace* of a transaction is a sequence of read and write requests it makes to the transaction manager. A history is written as a sequence of actions of the form $R_i(x)$ or $W_i(x)$, where $R_i(x)$ (or $W_i(x)$) means a transaction T_i issues a read (write) on data item x. Note that we are not interested in the values read or written, but in the syntactic properties of the string that lists the sequence of reads and writes on the data items.

We will assume at most one read and at most one write per transaction per data item in any trace. If a transaction reads as well as writes a particular data item, we assume the read will precede the write. Multiple reads and writes are handled in an obvious way: The first read is used to read the value of the data item and store it in local storage, and the other reads on the same data item are processed locally. Similarly, all writes except the last one are written to local storage, and the last one appears on the log. Thus there is no loss of generality.

A *log* of a transaction manager is a sequence of reads and writes granted by the transaction manager. As an example, three transaction traces and one possible transaction manager log are depicted below: (the notation is from Bernstein *et al.* [BeGo80].)

T_1 : $R_1(x)\ R_1(y)\ W_1(y)$
T_2 : $R_2(y)\ W_2(y)$
T_3 : $R_3(x)\ R_3(y)\ R_3(z)\ W_3(x)$

Log : $R_1(x)\ R_2(y)\ W_2(y)\ R_3(x)\ R_1(y)\ R_3(y)\ W_1(y)\ R_3(z)\ W_3(x)$

Note that when $A_n(x)$ precedes $A_n'(x')$ in some transaction trace then $A_n(x)$ has to precede $A_n'(x')$ in any log in which the transaction T_1 participates.

1.3 Increasing Concurrency.

In database concurrency control we are interested in protocols that maximize concurrency and work efficiently. However the 2-phase locking protocols may not score high in the concurrency criterion. 2-phase locking restricts the logs to a small subset of serializable logs. The restrictive nature of 2-phase locking is due to the fact that, it does not assume any *a-priori* knowledge of the intentions of the the transactions. Further, a 2-phase locking protocol can cause deadlocks, which further degrade its performance.

A 2-phase locking protocol has to lock all the necessary data items before it can unlock any. This is a strong constraint on the locking sequences that a

transaction may use, and may reduces concurrency in some cases. As a trivial example, consider the following transaction histories:

T_1: $R_1(x)$, $W_1(x)$, $R_1(y)$, $W_1(y)$
T_2: $R_2(x)$
T_3: $R_2(y)$

In this case, T_2 (and T_3) reads the value of x (and y), and does nothing else. Thus T_2 and T_3 can read x (or y) between any two actions of T_1, and still produce a correct serializable execution sequence. However if 2-phase locking is used, then T_2 (or T_3) is restricted to read x (or y) at only certain points, depending upon the locking sequence. For example suppose T_1 uses the following locking sequence: (LS(x) denotes setting a shared lock on x LX(x) denotes setting an exclusive lock on x, and U(x) denotes unlocking of x.)

x	y	
$LS_1(x)$		
$R_1(x)$		
$LX_1(x)$		
$W_1(x)$		↑
	$LS_1(y)$	
	$R_1(y)$	T_2 *cannot read x here.*
	$LX_1(y)$	↓
$U_1(x)$		
	$W_1(y)$	
	$U_1(y)$	

Trace of Transaction T_1.

This locking sequence prevents T_2 from reading x, between $W_1(x)$ and $R_1(y)$. However T_1's locking sequence can be changed so that T_2 can read x between $W_1(x)$ and $R_1(y)$, but then T_3 will not be able to read y between $W_1(x)$ and $R_1(y)$.

Since we *know* that T_2 and T_3 read x and y and do nothing else, we could state that T_2 and T_3 can read x and y interleaved between any steps of T_1. The situation would be different if T_2 or T_3 accessed or updated some other data items after reading x or y. However the 2-phase locking protocol does not rely upon, or have access to, information about the data access patterns of the transactions. We will

show how data access information can be used by concurrency control protocols to deal with situations like the one shown above.

The above example also shows that locking sequences of 2-phase locked transactions affect the amount of concurrency, depending on the transaction mix. A particular locking sequence in fact may favor one transaction over another. As there is no way to predict which transactions will run concurrently, it may not be easy to choose locking sequences. In fact, it is commonly believed that, locking should not be handled by transactions or application programs, but should be the responsibility of lower level, consistency preserving routines i.e. the transaction manager.

In order to make locking transparent, practical 2-phase locking schemes use read and write requests to obtain locks. A read or write request on an unlocked data item causes the lock to be obtained. All locks are released only when the transaction terminates. Thus it is intuitively clear that the locks are held for extended periods.

We propose a concurrency control protocol that, in cases like the above, would allow T_2 and T_3 to read x and y interleaved between any action of T_1. The locking would be handled entirely by the transaction manager. The protocol is inherently non-2-phase, and is relatively immune from deadlocks.

1.4 Related Work

It has been shown that some available information about the transactions or the database can be used for increasing concurrency. For instance the tree and DAG (directed acyclic graph) protocols can be used on databases structured like a tree or a DAG, respectively [SiKe80, KeSi82]. These protocols allow non-2-phase locking, and provide higher concurrency than 2-phase locking for transactions that traverse the tree or the DAG. In the DAG protocol a transaction is allowed to lock a child if it has locks on the majority of its parents (except for the first node locked), and unlocking may be done in any order. Thus the transaction can access only those data items that form a rooted subgraph of the original graph. These and similar protocols can be used in specialized applications and has practical limitations, but has received substantial theoretical interest.

If the writeset of a transaction is known in advance, timestamp protocols can be made abort free (or *progressive*) [BuSi83]. This can provide significant improvement in performance as aborts cause severe limitations of throughput in timestamp based systems.

Semantic knowledge about transaction actions has also been used to speed up transaction processing [Ga83]. A partial loss of serializability can be tolerated in some applications and can be used for better performance [FiMi82]. This may not be of interest in general purpose databases, where consistency is a major issue.

Our approach involves knowing in advance supersets of the readset and the writeset of the transactions. Knowing the exact readset (or writeset) of a transaction is better but not always feasible. However a superset of the readset (or writeset) can often be statically determined. We will assume this is possible, and demonstrate how this information can be used to achieve higher concurrency.

Read and write sets are used to control concurrency in SDD-1 (A System for Distributed Databases) [BeShRo80]. Transactions are divided into classes depending upon their read and write sets and then conflict graph analysis is performed. This is a *static* classification and is used to determine the nature of the conflicts. The conflict type is used to determine which protocol to use. (SDD-1 uses different protocols for different situations.) SDD-1 is a timestamp based system. In our approach the usage of read and write set information is dynamic. The protocol uses locking and static analysis of conflicting transactions is not performed.

2 The *Five Color* Protocol.

The *Five Color* Protocol is a non-two-phase locked protocol that ensures serializability in general (unstructured) databases. It derives its name from the five types of locks it uses.

A transaction T acts upon a set of data items D. A data item $x \in D$ is in the readset (Rd) of a transaction T (that is $x \in Rd(T)$) if the transaction does a read operation on x. Any item written by the transaction T is contained in the writeset (Wr) of T. Note that neither Rd(T) need be a subset of Wr(T) nor *vice versa*, and Rd(T) and Wr(T) may be supersets of the data items actually read and written by the transaction T.

Each transaction is required to declare its readset and writeset to the transaction manager before it issues any actions. The read and write sets can be parameters to the *begin-transaction* statement that is executed by a transaction when it starts. Since the readset (and writeset) are allowed to be supersets of the data items actually read (and written), they could be statically determined during query compilation. Sometimes the transaction may read and write different sets of data depending upon some statically undeterminable conditions. In this case the declared read and write sets should include all the items the transaction may act upon. However, for better performance the difference between the declared and actual read and write sets should be small.

2.1 The Basic Algorithm.

After the transaction manager knows the read and write sets of the transaction, it can obtain shared locks on all the data items in the readset, read them and store

the values in local storage. Then we would like to release the shared locks, as the locks seem no longer necessary. Also, we would also like to keep the data items in the writeset locked by a shared lock while the transaction is running, to prevent other transactions from updating them and causing missing updates. The shared lock on the writeset can then be upgraded to exclusive locks at commit time, and the values actually updated.

Thus our protocol is along the following lines:

- Get shared locks on the read and write sets.
- Read the readset into local storage and release the locks on the readonly items.
- Service the reads and writes issued by the transaction from and to local storage.
- Upgrade the shared locks on the writeset to exclusive locks and perform actual writes to the database during commit.

The merits of this protocol are the early release of shared locks and short holding of exclusive locks. The protocol is obviously non-2-phase, as some shared locks are released before some other shared locks are upgraded to exclusive locks. The problem with this initial scheme is that it can produce non-serializable schedules, and thus is unacceptable.

Now our aim is to use this basic idea, as described above, and add some checks to ensure serializability. We show that this is possible if we use some *marker locks* and a *validation phase*. The algorithms used to handle the marker locks are non-trivial and are described in detail in the following sections.

2.2 Locking.

The *Five Color* Protocol uses five types of locks, *White* (WL), *Blue* (BL), *Green* (GL), *Yellow* (YL) and *Red* (RL). The *White* and *Blue* locks are the marker locks[†]. They are compatible with all other locks (see Fig 1.). These are used by transactions to keep track of data items read or written by other transactions and cause *triggering* as described later.

The *Green* lock is a shared lock used for reading the readonly part of the readset.

[†] The reason for calling the *Blue* and *White* locks marker locks and not just markers are as follows. Markers are used by agents to mark objects. Irrespective of how many times an object is marked, it becomes unmarked when the marker is removed. With a lock we can ask questions such as *Which agents hold locks on this object?* or *What are the objects locked by this agent?* Conceptually markers are too weak to answer both these questions.

old → / new ↓	WHITE	BLUE	GREEN	YELLOW	RED
WHITE	Y	Y	Y	Y	Y
BLUE	Y	Y	Y	Y	Y
GREEN	Y	Y	Y	Y	N
YELLOW	Y	Y	N	N	N
RED	Y	Y	N	N	N

Fig 1: Lock Compatibility Table (Note: The table is asymmetric).

The *Yellow* lock is also a shared lock but is stronger than the *Green* lock. It is used to lock the items in the writeset, as a preparatory measure, before they are actually updated. The *Yellow* lock is compatible with the *Green* locks, allowing a *Yellow* locked item to be read as a readonly data item by another transaction, but it is not compatible with itself, preventing simultaneous update attempts. This basically prevent a deadlock condition, as two transactions, each trying to upgrade a shared lock on the same item to an exclusive lock, cause the simplest deadlock situation. We could allow the *Yellow* lock to be compatible with itself (making it a true shared lock) but this would serve no purpose (except increase chances of deadlocks).

Note that a *Green* lock can be obtained on a *Yellow* locked item but a *Yellow* lock cannot be obtained on a *Green* locked item. This feature makes the compatibility matrix asymmetric. This asymmetry is not a major feature of the algorithm but has to be provided to prevent a particular race condition that can arise in the lock acquisition phase, described later.

The *Red* lock is the exclusive lock used for writing, and is compatible only with the *White* and *Blue* marker locks. Neither the *Green* locks nor the *Red* locks are held over extended lengths of time. Only *Yellow*, *Blue* and *White* locks exist nearly as long as the transaction does.

2.3 Transaction Phases

A transaction T goes through several phases. When the transaction is initiated, (*arrival point*), the transaction manager obtains *Yellow* locks on the data items in Wr(T) and *Green* locks on Rd(T) − Wr(T) i.e. the *read-only* data items. This is the *lock acquisition phase*.

After all the locks are obtained, the transaction is *validated*. If the transaction passes validation then it has to acquire some *Blue* and *White* locks. It gets *Blue* (and *White*) locks on data items written (and read) by some other concurrently running transactions. It also has to donate *White* and *Blue* locks on the items in its read and write sets to some other transactions. This is called *lock inheritance phase*, and the exact details of which data items are locked are explained later. After completion of the lock acquisition phase, the transaction reaches its *locked point*.

Subsequently all the items in Rd(T) are read into local storage, the *Green* locks are converted (downgraded) to *White* locks and the transaction enters the *processing phase*. Then the transaction commences execution (*start point*). When the transaction completes execution it commits (*commit point*), and all *Yellow* locks are converted to *Red* locks, the updated items are written to the database, all locks are released, and the transaction terminates. These phases and points are illustrated in Fig. 2.

The following is an informal outline of how a transaction manager handles a transaction.

→ *Arrival Point* (Transaction T arrives)

- Get *Yellow* locks on Wr(T) and compute Before/After sets (explained later)
- Get *Green* locks on Rd(T) − Wr(T)
- Do validation and lock inheritance processing (explained later)

→ *Locked Point*

- Read values of Rd(T) into local storage,
- Downgrade *Green* locks to *White* locks,
- Start transaction processing.

→ *Start Point*

- Let T commence processing,
  if T issues read(x), then return the value of x from local storage,
  if T issues write(x), then write x in local storage.

→ *Commit Point*

- Upgrade *Yellow* locks to *Red* locks,
- Write updated items to the database,
- Release all (*White*, *Blue*, and *Red*) locks held by T.

→ *Termination Point* (Transaction T terminates)

2.4 Transaction Manager Algorithms.

A transaction is said to be *live* if it has arrived, but has not terminated. For each live transaction T, the Transaction Manager maintains two temporary sets, during lock acquisition phase, called Before(T) and After(T). These sets are accessed and updated only during the lock acquisition phase. The set Before(T) is a set of transaction that are live, conflict with T, and must come before T in a serialization order. Similarly After(T) contains those live and conflicting transactions that must come after T in the serialization order. The serialization order is determined by the actual order in which the locks are requested by the concurrent transactions. (Sometimes the Before and After sets contain a few recently terminated transactions, but that is of no major consequence.)

As the transaction manager acquires locks on behalf of a transaction, it can determine which transactions must come before or after this transaction in the serialization order, by looking at the existing locks. These transactions are placed in the Before(T) and After(T) sets. The Before(T) and After(T) sets are constructed as follows.

Suppose T wants a *Green* lock on a data item x, and a set of transactions $\{T_i, T_j, \ldots, \}$ already possess *Blue* locks on x. The existence of *Blue* locks held by $\{T_i, T_j \ldots, \}$ implies that some transaction(s) later than all of $\{T_i, T_j \ldots, \}$ have written x. As T wants to read x, it must come after all of $\{T_i, T_j, \ldots, \}$. Thus $\{T_i, T_j, \ldots, \}$ must logically precede T, and they are added to Before(T).

However, if some transaction T_i is holding a *Yellow* lock on x (when T is trying to *Green* lock it), this implies T_i will update x after T reads it. Hence T_i should come after T, and T_i is added to After(T).

Similarly, during an attempt to *Yellow* lock an item x, all transactions holding *Blue* or *White* locks on x are added to Before(T).

Validation is simply checking whether Before(T) $\cap$ After(T) = $\varnothing$. If not, this implies that there are transactions that must come before as well as after T in the serialization order, and thus the resulting execution would be non-serializable. To prevent this from occurring, the transaction T is rescheduled. Avoiding livelocks (or starvation) due to rescheduling is dealt with in section 6.

If the transaction passes validation, then the *lock inheritance processing* has to be done. Some *Blue* and *White* locks are granted to various transactions as a result of lock inheritance. This is done as follows.

Suppose T has passed validation. T is delayed until all the transactions in After(T) reach their respective locked points.) Then T is given *White* locks on all the data items *White* locked by transactions in After(T), and *Blue* locks on all the data items *Blue* locked by transactions in After(T). Then all transactions in Before(T) are given *White* locks on the readset of T, *Blue* locks on the writeset of T. Finally, all transactions in Before(T) get *White* locks on all data items *White* locked by T, and *Blue* locks on all data items *Blue* locked by T. Note that during this last two steps of the lock acquisition phase of transaction T, some transactions other than T get some *Blue* and *White* locks. (These other transactions are the transactions in Before(T).) After lock inheritance, the transaction actually starts executing, entering its processing phase.

The algorithms stated above are formally restated in pseudo Pascal. Some of the abbreviations used are:

WL → *White* Lock
WL-ed → *White* Locked
WLS(T) → Set of data items *White* locked by T
LOCK(*White*,x) → Obtain a *White* lock on data item x. If lock unavailable due to a conflict, wait until it can be obtained.

(similar, for all other colors)

i) Getting *Yellow* Locks:

```
Before(T) ← ∅;
for all x ∈ Wr(T) do
   begin
     LOCK (Yellow, x);
     Before(T) ← Before(T) ∪ { T_i | x is WL-ed or BL-ed by T_i}
   end;
```

ii) Getting *Green* Locks:

```
After(T) ← ∅;
for all x ∈ (Rd(T) − Wr(T)) do
   begin
     LOCK (Green, x);
     Before(T) ← Before(T) ∪ { T_i | x is BL-ed by T_i };
     After(T) ← After(T) ∪ {T_i | x is YL-ed by T_i}
   end;
```

iii) Validation and lock inheritance:

```
if (After(T) ∩ Before(T) ≠ ∅)
        (* Validation *)
then RESCHEDULE T
else
  begin
    for all τ ∈ After(T) do
        begin
          Wait for τ to reach locked point;
            BLS(T) ← BLS(T) ∪ BLS(τ) ∪ Wr(τ);
                (* T gets Blue locks on the writeset of τ
                    and all items Blue locked by τ *)
            WLS(T) ← WLS(T) ∪ WLS(τ) ∪ Rd(τ)
                (* T gets White locks on the readset of τ
                    and all items White locked by τ *)
        end;

     for all τ ∈ Before(T) do
        begin
          BLS(τ) ← BLS(τ) ∪ BLS(T) ∪ Wr(T);
                (* τ gets Blue locks on all items
                    Blue locked by T and the writeset of T *)
            WLS(τ) ← WLS(τ) ∪ WLS(T) ∪ Rd(T)
                (* τ gets White locks on all items
                    White locked by T and the readset of T *)
        end
  end;
```

We stress that there is no assumption of atomicity of any part of the above Transaction Manager algorithms, except in the test and set needed for the implementation of the LOCK function. The LOCK function is the standard locking primitive. If the lock requested cannot be granted due to a conflict, then the process is suspended (or queued) until the lock can be granted. These algorithms can be executed concurrently with all the activities of the other transaction on the database system, including lock acquisition phases of other transactions.

In our protocol, locks can be held only by live (or uncommitted) transactions. Hence if a transaction τ in Before(T) commits before T gets to do lock inheritance,

then τ does not have to be given any locks.

2.5 Intuitive Discussion.

The transaction manager of the *Five Color* protocol handles all the locking and concurrency control needed to run transactions. The transactions themselves do not have any knowledge of the protocols. A transaction has the following structure:

```
Begin-Transaction ( Readset, Writeset)
    ...
    { Statements }
    ...
End-Transaction.
```

The Begin-Transaction statement starts up the lock acquisition phase of the transaction. The transaction manager does the acquiring of locks (using the readset and writeset information provided as parameters) and then performs the validation and inheritance phases. After all the preprocessing is completed, the transaction starts executing the statements, needing no further assistance from the transaction manager. When the End-Transaction statement is reached, the transaction manager is called upon to do the *Yellow* to *Red* lock upgrades, actual writes and commit.

Though acquiring locks are done on behalf of the transaction, by the transaction manager, in the rest of our discussions we will refer to this event as "a transaction obtains a lock" because of simplicity and conceptual clarity.

As described in section 2.1, the basic algorithm that the *Five Color* Protocol uses is as follows. First, the writeset is locked using the *Yellow* lock. The *Yellow* lock is the shared lock that allows reading, but is not compatible with itself. This allows reading of to-be-updated items, but does not allow missing updates or simple deadlocks (section 2.2).

Then the readset is locked with *Green* locks. The *Green* lock is a shared lock (or read lock). After the locking of the readset, all the items in the readset is read into local storage, and is available to the transaction when it needs to actually read them. Then the *Green* locks are downgraded to *White* locks.

After the locks are obtained, the validation and lock inheritance processing is done, the transaction commences execution and finally commits. In the commit phase, the *Yellow* locks are upgraded to *Red* locks, the data written out, and all locks released. The need for validation, and how it is done is discussed later in this section.

The properties that allow more concurrency in the *Five Color* protocol are the early release of all read locks (*Green* locks), and the shared nature of the *Yellow* lock.

Suppose Transaction T_1 is running, and it has released all the *Green* locks on the readset. Now T_2 can update an item which was read by T_1, making T_2 a logically *later* transaction in the serialization order. In 2-phase locking T_2 has to wait until T_1 releases the read lock, and this may mean waiting until T_1 commits. If there are many reading transaction on database systems, transaction that write can be held up for long periods of time.

In addition, write locks held by transactions, updating data items, cause delays for reading transactions. The *Yellow* locking scheme allows readonly transactions to read data items even in the presence of update transactions.

The early release of read locks may allow non-serializable executions, and this has to be avoided. Validation is used to avoid possible inconsistencies. The following example shows the need for validation.

Suppose T_1 is running and has read x and released the *Green* lock on x. T_1 has a *Yellow* lock on y which it intends to update later. Now a new transaction T_2 can choose to update x (get a *Yellow* on x), making T_2 a logically later transaction than T_1. T_2 can also attempt to read (or get a *Green* lock) on y, an item that is *Yellow* locked by T_1 (making T_2 come logically before T_1). This would lead to a non-serializable condition

This is a simple case of a possible non-serializable execution that may occur. It may seem this is a easy condition to detect without going into the complexities of validation and lock inheritance as described in previous sections. However, there are situations involving several transactions and several data items that turn out to be too complex to detect, and the validation scheme described is one of the simplest schemes that detect them. The validation works as follows.

A transaction T_1 holds *White* locks on all items it has read. T_1 also holds *White* locks on data items read by other transactions that (are active[1] or were active during the lifetime of T_1, and (ii) should come **after** T_1 in the serialization order. Similarly, T_1 holds *Blue* locks on all items written (or to be written)[2] by transactions that were active in T_1's lifetime and should come **after** T_1 in the serialization order. (This property is proved later in Lemma 4). This implies T_1 "knows" about all the data items read or written by "later" transactions.

[1] A transaction is *active* if it has reached locked point, but has not started acquiring *Red* locks.

[2] The fact that T_1 holds *Blue* locks on data items that have not yet been updated can lead to an anomaly in certain situations. See section 5 for a modification that avoids this.

Suppose a new transaction T_2 arrives in this situation, and *Yellow* locks an item that is *Blue* or *White* locked by T_1. This implies T_2 will update an item that has been read or written by some transaction that is after T_1 in the serialization order. Thus T_2 should be after T_1 in the serialization order. As expected, the *Yellow* locking of this item causes T_1 to be placed in Before(T). before T_2 in the serialization order. In fact all transactions such as T_1 which should come before T_2 get into Before(T_2).

Now T_2 can cause a non-serializable condition by attempting to *Green* lock an item, which is already *Yellow* locked by a transaction such as T_1. *Green* locking a *Yellow* locked item implies that T_2 is getting a view of the database as it was before T_1 updates the item it *Yellow* locked, thus T_2 should precede T_1. But this action would cause T_1 to get into After(T_2), and T_2 would fail validation.

The fact that each transaction T_1 has *White* and *Blue* locks on the read and write sets of all transactions that come after T_1 is ensured as follows. Whenever a new transaction, T_2, determines all the active transactions which follow it (by means of the After(T_2) set), it acquires *White* (and *Blue*) locks on the readset (and writeset) of all transaction in After(T_2). Similarly, it gives all the transactions in Before(T_2) *White* locks on all its *White* locked objects, and *Blue* locks on all its *Blue* locked objects and Wr(T_2).

The maintenance of the *White* and *Blue* locks might seem contrived and unnecessary, but it has to be done to prevent conditions typified by the following example.

T_1	T_2	T_3	Notes
Arrives, *Yellow* locks y *Green* locks x reads x Converts *Green* on x to *White*.			
Gets *Blue* lock on z	Arrives, *Yellow* locks x *Yellow* locks z Updates x Updates z terminates		$T_1 \in$ Before(T_2)
		Arrives *Yellow* locks z *Green* locks y	$T_2 \rightarrow T_3$ $T_3 \rightarrow T_1$ (...interesting part, see below.)

If T_3 were allowed to continue, then it would read y and write z, and cause a non-serializable execution. Note that at this point T_2 is no longer alive, and T_1 does not touch z. So there seems to be no basis for disallowing T_3 from reading y and writing z, unless if we use the *Blue* and *White* locks, and the Before and After sets.

When T_3 *Yellow* locked z, z was *Blue* locked by T_1. This caused T_1 to become a member of Before(T_3). When T_3 *Green* locked y, since y was *Yellow* locked by T_1, T_1 became a member of After(T_3). Now that the intersection of Before(T_3) and After(T_3) is not empty, T_3 fails validation, and the non-serializable execution is not allowed.

2.6 An Example

Consider the following log:

Log $R_1(x)\ R_2(y)\ W_1(y)\ R_3(z)\ W_2(z)$
Serial order: $T_3 \rightarrow T_2 \rightarrow T_1$

The log is non-2-phase locked as may be seen by inspection. This log is allowed by the *Five Color* protocol. The behavior of the protocol in response to the above log is depicted in Fig. 3.

The above log has an interesting property. Note that although T_1 completes execution before T_3 commences, T_3 precedes T_1 in the serialization order. This violates a property of some serializable logs called *strict serializability*[†] [BeGo80]. Thus the above log is classified as a *non-strictly-serializable*.

Most well known locking protocols do not allow non-strictly-serializable logs. That is the logs allowed by these protocols are proper subsets of strictly serializable logs. As 2-phase locking does not allow such logs, the above log cannot be allowed by any 2-phase locking scheme. This demonstrates that the ability of the *Five Color* protocol extends beyond the scope of 2-phase locking.

3 Properties and Correctness:

We now state the properties of the *Five Color* protocol and prove it achieves serializability. The following Lemmas are useful to understand the details of the algorithms, but the actual proofs may be skipped at the first reading. In order to show that the *Five Color* protocol assures serializability, we define a standard precedence relation $\rightarrow$. This relation is created amongst transactions as the *Five Color* scheduler executes in a manner consistent with the conflicts that occur amongst the transactions. We show that this relation, for transactions under the protocol is cycle free. All transactions that partake in the relations in the definitions are assumed to be transactions that have already passed the validation phase. Transactions that have not passed validation also generate relationships with other transactions, but since these transactions do not contribute anything to the processing environment, we choose to ignore them in the correctness proofs.

Definitions:

Define a binary relation ($\rightarrow$) over transactions. Two transactions T_i and T_j are related by the precedence relation ($T_i \rightarrow T_j$), if and only if:

[†] A log L is strictly serializable if there exists a serial order L_s of L such that if T_1 and T_2 are in L and their action do not interleave, then T_1 and T_2 appear in the same order in L_s as in L.

i) T_i reads some data item x, and at some later point T_j writes x (r-w conflict), or

ii) T_i writes some data item x, and at some later point T_j reads x (w-r conflict), or

iii) T_i writes some data item x, and at some later point T_j writes x (w-w conflict).

Lemma 1

The relation $T_i \rightarrow T_j$ occurs if and only if one of the following cases take place:

i) T_i gets a *Green* lock on x and then T_j gets a *Yellow* lock on x after T_i releases the *Green* lock. This arc is defined as of type $[\alpha \rightarrow \beta \mid \alpha G, \beta Y]$.[†]

ii) T_j gets a *Yellow* lock on x, then while T_j holds the *Yellow* lock, T_i gets a *Green* lock on x, and then T_j converts the *Yellow* lock into a *Red*. This arc is defined as of type $[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$.

iii) T_i gets a *Yellow* lock on x, and later, after T_i unlocks x, T_j gets a *Green* lock on x. This arc is defined as of type $[\alpha \rightarrow \beta \mid \alpha Y, \alpha R, \beta G]$.

iv) T_i *Yellow* locks x, and later T_j *Yellow* locks x. This arc is defined as of type by $[\alpha \rightarrow \beta \mid \alpha Y, \beta Y]$.

Proof:

Simple, but lengthy.

[]

The two cases ii) and iii) (above) look similar but cause very different results. In ii), T_j gets a *Yellow* lock on x and then, while the *Yellow* lock is held, T_i gets a *Green* lock on x. In this case, T_j will update x, but T_i gets to see x as it existed before T_i updated it. Hence the serialization order of the transactions should be $T_i \rightarrow T_j$. However in case iii), under similar circumstances, if T_j gets a *Yellow* lock, and after this *Yellow* lock has been converted to *Red*, and released, T_j gets a *Green* lock on x, then the serialization order should obviously be $T_j \rightarrow T_i$. (The roles of T_i and T_j in the second example has been reversed, to be similar to the first example.)

The arcs of the precedence relation graph ($\rightarrow$) are caused by r-w, w-r and w-w conflicts. These conflicts happen when both the transactions have actually processed the conflicting reads and writes. However for ease of modeling we will assume that the arcs are created earlier. We define that an arc $T_i \rightarrow T_j$ is *created* when either T_i

[†] The notations such as $[\alpha \rightarrow \beta \mid \alpha G, \beta Y]$ denotes *types* of edges. If two transactions T_1 and T_2 are related as $T_1 \rightarrow T_2$, this relation can be caused in many ways. α stands for the transaction to the left of the $\rightarrow$ symbol, and β is the transaction on its right. R,Y and G denote getting a *Red Yellow* or *Green* lock. The sequence to the left of the '|' shows the sequence of getting locks by α and β which led to the formation of the $\rightarrow$ relation.

or T_j reaches locked point, whichever is later. (T_i and T_j must of course conflict as stated in Lemma 1.) This occurs after all the locks for T_i and T_j have been obtained, but before any actual conflict due to actual reading or writing has taken place. Also, if a transaction has not reached locked point, there is no arc either from or to it.

Note that the precedence graph thus formed is not identical to the graph formed by the r-w, w-r and w-w conflicts at some given point in time. This is because the arcs of the precedence graph are created prior to occurrence of the conflicts. Thus this precedence graph is in fact a superset of the conflict graph, which will become identical to the conflict graph when all activity on the database ceases. However our correctness criterion is the acyclicity of the conflict graph (see below) and acyclicity of the precedence graph implies acyclicity of the conflict graph.

Lemma 2

If $T_i \rightarrow T_j$ and T_j reached its *locked point* before T_i did, then the arc can only be of type $[\alpha\rightarrow\beta \mid \beta Y, \alpha G, \beta R]$.

Proof:

Since T_j has reached its *locked point*, it has obtained all its locks. The arc $T_i \rightarrow T_j$ could have been caused by four cases (Lemma 1). Consider each case separately:

$[\alpha\rightarrow\beta \mid \alpha G, \beta Y]$

After T_i gets a *Green* lock on x, T_j cannot get a *Yellow* lock on x until T_i converts the *Green* lock to a *White* lock. Hence T_j cannot reach the *locked point* before T_i does and this case is impossible.

$[\alpha\rightarrow\beta \mid \beta Y, \alpha G, \beta R]$

This case is possible, as T_i may obtain a *Green* lock on x after T_j has placed a *Yellow* lock on x.

$[\alpha\rightarrow\beta \mid \alpha Y, \alpha R, \beta G]$

In this case T_i has to get a *Red* lock on x before T_j gets a *Green* lock on x. This makes it impossible for T_j to get to the *locked point* before T_i, thus this case is impossible.

$[\alpha\rightarrow\beta \mid \alpha Y, \beta Y]$

Again, T_i has to release the *Yellow* lock on x before T_j can place another *Yellow* lock on x. Hence T_j cannot reach its *locked point* before T_i, making this

case impossible too.

Thus only the second case can take place, and hence the arc is of type $[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$.

[]

Note, that as defined earlier, a transaction is *active* if it has reached locked point, but has not started acquiring *Red* locks.

Lemma 3

If $T_i \rightarrow T_j$ and T_j reached its *locked point* before T_i did, then T_j is still *active* when T_i reaches its *locked point*.

Proof

The arc $T_i \rightarrow T_j$ is of the type $[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$ (Lemma 2). Thus T_j gets a *Yellow* lock first, then T_i gets a *Green* lock and finally T_j upgrades its *Yellow* lock to *Red*. After T_i gets a *Green* lock on x, T_j cannot convert its *Yellow* lock to a *Red* lock until T_i converts its *Green* lock to a *White* lock. Thus when T_i have reached *locked point*, T_j must be active.

[]

Lemmas 2 and 3 show that unlike 2-phase locking the precedence graph can grow "backwards" (see section 5). A transaction T_2, which arrives later than transaction T_1, may precede T_1 in the precedence order, if T_1 is active when T_2 arrives.

Lemma 4

If $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$ is a path in the precedence relation, and T_1 is active, then

i) T_1 possesses *White* locks on $Rd(T_n)$ (i.e. $Rd(T_n) \subset WLS(T_1)$), and
ii) T_1 possesses *Blue* locks on $Wr(T_n)$ (i.e. $Wr(T_n) \subset BLS(T_1)$).

Proof:

i) Proof is by induction on n, the number of transactions in the path.

$\boxed{n = 2}$

Let $T_1 \rightarrow T_2$, where T_1 is active. The arc in the path can be of four types. Consider each case separately:

$[\alpha \rightarrow \beta \mid \alpha G, \beta Y]$

T_2 can obtain the *Yellow* lock on the data item (say x) after T_1 has converted

the *Green* lock it was holding to a *White* lock. When T_2 gets a *Yellow* lock on x when T_1 holds a *White* lock on it, T_2 is added to Before(T_2). Then T_1 inherits *White* locks on T_2's readset. Hence $Rd(T_2) \subset WLS(T_1)$.

[α→β | βY,αG,βR]

When T_1 gets a *Green* lock on x while T_2 is holding a *Yellow* lock on x, T_2 gets into the set After(T_1). Then T_1 inherits *White* locks on the readset of T_2. Hence $Rd(T_2) \subset WLS(T_1)$.

[α→β | αY,αR,βG]

T_1 cannot be active in this case.

[α→β | αY,βY]

T_1 cannot be active in this case.

Thus T_1 has *White* locks on the readset of T_2.

ii) Similar. Substitute *Blue* for *White* and writeset for readset in above to show that T_1 has *Blue* locks on the writeset of T_2.

n > 2

Assume i) and ii) are true for all paths consisting of up to n-1 transactions. Consider a path consisting of n (n>2) transactions. By definition all the transactions in this path have reached their *locked points*. Let T_k (1<k<n) be the transaction that reached its locked point last, amongst all the transactions in the path:

$$T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_{k-1} \rightarrow T_k \rightarrow T_{k+1} \rightarrow T_{k+2} \rightarrow \ldots \rightarrow T_n$$

Consider the instance when T_k just reached its *locked point*. At this point the arcs:

$T_{k-1} \rightarrow T_k$ and
$T_k \rightarrow T_{k+1}$

are created. Thus prior to this there were two shorter paths,

a) $T_1 \rightarrow \ldots \rightarrow T_{k-1}$ and
b) $T_{k+1} \rightarrow \ldots \rightarrow T_n$.

For path a), T_1 is alive (by assumption) and thus T_1 has *White* locks on $Rd(T_{k-1})$ and *Blue* locks on $Wr(T_{k-1})$. As T_k reached *locked point* later than T_{k+1}, it follows

that T_{k+1} was active when T_k reached *locked point*, and thus had *White* locks on $Rd(T_n)$, and *Blue* locks on $Wr(T_n)$ (because of path b).

By Lemma 2, the arc $T_k \rightarrow T_{k+1}$ must be of type $[\alpha\rightarrow\beta \mid \beta Y, \alpha G, \beta R]$. Hence T_k gets a *Green* lock on some x, that is *Yellow* locked by T_{k+1}. Thus T_{k+1} is in $After(T_k)$, and since T_k gets *White* locks on all items *White* locked by T_{k+1}, T_k gets *White* locks on $Rd(T_n)$.

The $T_{k-1} \rightarrow T_k$ arc can be of four types. Treating them separately:

$[\alpha\rightarrow\beta \mid \alpha G, \beta Y]$

In this case T_k sets a *Yellow* lock on a data item x, which is in $Rd(T_{k-1})$, and thus x is *White* locked by T_1. This makes $T_1 \in Before(T_k)$, and thus T_1 inherits its *White* locks on all $WLS(T_k)$, which contains $Rd(T_n)$. Thus T_1 gets *White* locks on $Rd(T_n)$.

$[\alpha\rightarrow\beta \mid \beta Y, \alpha G, \beta R]$

This cannot happen as T_{k-1} reaches *locked point* before T_k. In other words, if T_k gets a *Yellow* lock, and then T_{k-1} gets a *Green* lock, T_k becomes a member of the set $After(T_{k-1})$. Now T_{k-1} has to wait for T_k to reach locked point, before it can reach locked point. (This is a condition during lock inheritance, please see the algorithm description on page 12.)

$[\alpha\rightarrow\beta \mid \alpha Y, \alpha R, \beta G]$

In this case T_k sets a *Green* lock on a data item x, which is in $Wr(T_{k-1})$, and thus x is *Blue* locked by T_1. This causes $T_1 \in Before(T_k)$, and thus T_1 inherits *White* locks on all $WLS(T_k)$, which contains $Rd(T_n)$. Thus T_1 gets *White* locks on $Rd(T_n)$.

$[\alpha\rightarrow\beta \mid \alpha Y, \beta Y]$

In this case T_k sets a *Yellow* lock on a data item x, which is in $Wr(T_{k-1})$, and thus x is *Blue* locked by T_1. This makes $T_1 \in Before(T_k)$, and thus T_1 inherits *White* locks on all $WLS(T_k)$ which contains $Rd(T_n)$. Thus T_1 gets *White* locks on $Rd(T_n)$.

Thus for all cases, T_1 has *White* locks on readset of T_n.

ii) Similar. Substitute *Blue* for *White* and writeset for readset in above to show that T_1 has *Blue* locks on the writeset of T_n.

[]

Lemma 4 shows the most important property of the *Five Color* protocol. This implies that if a transaction T_1 is active, it "knows" about the read and write set of

all transactions that come after T_1. This property is used to achieve serializability by causing a validation conflict when a cycle is created by some transaction.

Theorem 1:

The protocol ensures serializability.

Proof

We show that the precedence graph is acyclic. The proof is by contradiction. Assume there can be a cycle in the $\rightarrow$ relation. Choose a minimal cycle:

$$T_1 \rightarrow T_2 \rightarrow \ldots \ldots \rightarrow T_{k-1} \rightarrow T_k \rightarrow T_1$$

Assume that T_k is the transaction to reach its *locked point* last, compared to all the other transactions participating in the cycle. It will be shown that if T_k accessed data items in a manner that caused the cycle in the $\rightarrow$ relation, then T_k would have been rescheduled at the validation phase.

T_k causes the creation of two arcs, which cause the cycle. They are the arcs:

A1: $T_{k-1} \rightarrow T_k$ and

A2: $T_k \rightarrow T_1$.

As T_k is the last transaction to reach its *locked point*, T_1 must have reached its *locked point* before T_k, hence (by Lemma 2) the arc $<T_k \rightarrow T_1>$ must be of type $[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$. Thus T_k gets a *Green* lock on a data item *Yellow* locked by T_1. Hence $T_1 \in After(T_k)$

Also, by Lemma 3, T_1 must have been active when T_k reached its *locked point*. Thus by Lemma 4, T_1 has *White* locks on $Rd(T_{k-1})$ and *Blue* locks on $Wr(T_{k-1})$.

The arc $<T_{k-1} \rightarrow T_k>$ could be of four types. Let us treat them separately:

$[\alpha \rightarrow \beta \mid \alpha G, \beta Y]$

In this case $T_1 \in Before(T_k)$ (see proof of Lemma 4). Thus $T_1 \in (Before(T_k) \cap After(T_k))$, and hence T_k should have been rescheduled at validation, and the cycle could not have resulted.

$[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$

This type of arc could have been caused if and only if T_k reached *locked point* before T_{k-1}, which is not the case.

$[\alpha \rightarrow \beta \mid \alpha Y, \alpha R, \beta G]$

In this case $T_1 \in Before(T_k)$ (see proof of Lemma 4). Thus again $T_1 \in (After(T_k) \cap Before(T_k))$ and T_k should have been rescheduled.

$[\alpha \rightarrow \beta \mid \alpha Y, \beta Y]$

In this case, again $T_1 \in Before(T)$. Rest as above.

Thus there can be no cycle in the → relation under the protocol, and hence the protocol ensures consistency of updates.

[]

4 A Modification

The algorithm as described has an undesirable feature. It does not hamper consistency but may cause more aborts than necessary. Consider the following situation:

1) T_1 gets *Green* lock on x
2) T_1 downgrades *Green* lock to *White* lock
3) T_2 gets *Yellow* lock on x
4) T_3 gets *Green* lock on x

At step 3, T_1 is in Before(T_2) and thus gets a *Blue* lock on x. When T_3 gets a *Green* lock on x, Before(T_3) contains T_1. Actually T_1 and T_3 are unrelated. However as T_1 believes T_3 wrote x after T_1 read it thus any transaction reading x should come after T_1. The anomaly exists as T_2 has *not yet written x* when T_3 reads x. There would have been no problem if T_2 had terminated before T_3 read x, and in this case T_3 would have to be after T_1.

Thus in a path of transaction $T_1 \rightarrow T_2, \ldots, \rightarrow T_k$, T_1 should have *Blue* locks only on those data items that have been updated by $T_2, \ldots, T_k$, and not on all data items in the writeset of $T_2, \ldots, T_k$.

The following is a very brief description of a method to prevent the above anomaly. Introduce another type of lock, called a I-*Blue* lock (or Intent-*Blue* lock). During lock inheritance, *Blue* locks are obtained on items already *Blue* locked by other transaction, while I-*Blue* locks are obtained on the writesets (uncommitted updates) of the transactions concerned. When a transaction commits, all the I-*Blue* locks held on its writeset by other transactions are changed to *Blue* locks. All other algorithms remain the same.

This modification is not incorporated in the algorithm as described in section 2. This is to avoid introduction of extra complexity, which has no bearing on the correctness of the protocol, and simplify understanding of the protocol. This is not a correctness issue, nor an important point in the concepts used in the *Five Color* protocol.

5 Deadlocks

In the *Five Color* protocol there is potential for two types of deadlocks, one due to locking and the other due to waiting for other processes to reach their locked

points. Let us address them separately.

The first form of deadlock is the one encountered in traditional locking protocols, and is caused by transactions in a circular wait, trying to obtain locks. This form of deadlock can be prevented in this protocol. Since we know beforehand the resources (locks) that the transaction needs to access. We define an ordering of resources and locks are obtained in an increasing order. First all the *Yellow* locks are obtained, in increasing order, and then all the *Green* locks are obtained in that order. The *Yellow* to *Red* conversion is also done in the increasing order of resources (data items).†

Theorem -

The pre-ordered locking strategy used in the *Five Color* protocol is deadlock free.

Proof

Suppose deadlocks can take place. Consider an instance of a deadlock involving k transactions, with the cycle in the waits for graph being:

$$T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k \rightarrow T_1$$

There are two cases.

i)

Suppose none of the transactions T_1 to T_k are in the process of acquiring *Red* locks, that is they are in the *Green* or *Yellow* locking phase. Now T_1 to T_k cannot be all in the *Green* lock or *Yellow* lock acquiring phase. (If there is one type of locking, this strategy is known to be deadlock free). Thus some transactions are waiting for a *Green* lock, and others are waiting for a *Yellow* lock. Hence at some point, a transaction T_i is waiting for a *Green* lock on a data item x, for which T_j is holding a *Yellow* lock. This is a contradiction as the *Green* lock is compatible with an existing *Yellow* lock.

ii)

† If we use only one type of (exclusive) lock, this strategy of obtaining locks in a predefined fashion is known to be deadlock free. However it is not true in general, where several types of locks with various compatibilities are used. In our case, however, it is deadlock free, as shown in theorem 2.

Suppose at least one of the transactions, say T_1, is in the process of upgrading its *Yellow* locks to *Red* locks. A transaction which is trying to upgrade a *Yellow* lock on x, to a *Red* lock will have to wait only if some other transaction holds a *Green* lock on x (as no other transaction can hold a *Yellow* lock on x). Thus if T_1 is waiting for T_2, then T_2 must be in its *Green* lock acquiring phase.

Similarly, a transaction trying to acquire a *Green* lock on x will wait for another transaction only if that transaction holds a *Red* lock on x, as *Red* is the only lock incompatible with the *Green* lock.

Thus if $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k \rightarrow T_1$ is a cycle of waiting transactions and T_1 is in the *Red* lock acquiring phase, then so is T_3, T_5 ..., and T_2, T_4 ..., are in the *Green* lock acquiring phase (that is, the cycle comprises **only** of transactions in the *Green* and *Red* lock acquiring phases.

The proof that there cannot be a cycle in a set of transactions having the above properties, is very similar to the proof that there cannot be a cycle in a set of transactions acquiring exclusive locks in a predefined order, and is not included here for brevity.

[]

The second form of deadlock is peculiar to this protocol. This form of deadlock involve one or more transactions in the lock inheritance phase. Note that in the lock inheritance phase, a transaction T has to wait for all transactions in After(T) to reach locked point. This can cause deadlocks. The following is an example of a deadlock caused by two transactions in the lock inheritance phase.

i) T_1 *Yellow* locks x
ii) T_2 *Yellow* locks y
iii) T_1 *Green* locks y
iv) T_2 *Green* locks x

In this sequence of events the following problem takes place. Steps i) and ii) cause no surprises, but in Step iii) as T_1 tries to *Green* lock y, which is *Yellow* locked by T_2, T_2 becomes a member of After(T_1). Similarly, when T_2 *Green* locks x, T_1 becomes a member of After(T_2).

Now both transactions will pass the validation, and wait for all transactions in their After sets to reach locked point. T_1 will thus wait for T_2 to reach locked point before it can reach locked point, and vice versa. Thus we have a deadlock. This deadlock effectively prevents the non-serializable log that could result if the transactions were allowed to continue.

The deadlock can involve transactions waiting for locks as well as transactions waiting for other transactions to reach locked point. However as proved above there are no deadlocks involving only transactions waiting for locks.

The deadlocks can be detected by standard deadlock detection algorithms, or by timeouts. As the deadlock occurs before the transactions start execution, there is no rollback involved and the overhead suffered is small. Also the probability of deadlocks seems to be lower than 2-phase locking. The reasons being absence of deadlocks due to locking and lower population of transactions that can participate in deadlocks. (Only transactions in the pre-processing phase can participate in deadlocks.)

Since the transactions which participate in deadlocks do not do any processing, we believe timeouts may be an easier and more efficient method to deal with deadlocks. Lack of processing means there are no processing delays and the timeouts can be fine tuned better to take into account the locking delays and cause deadlock warnings if something takes too long to happen.

Every deadlock cycle has at least one transaction waiting for the completion of the lock acquisition phase of another transaction. We propose that this is the point where a timeout should be introduced. The delay for the timeout can be a function of the number of locks the other transaction has to acquire. This will keep the probability of detection of false deadlocks to be low.

6 Livelocks

There is a small chance of livelocks or starvation in this protocol. This is due to the rescheduling of transactions because of validation failure. There is no guarantee that an aborted transaction will finally be able to run. However we feel that the chances of starvation is extremely small. But low probability of starvation is still not a guarantee against starvation, so we propose the following method of avoiding livelocks.

If a transaction get aborted due to validation failure a large number of times, we label the transaction as *starvation prone*. The writeset of a starvation prone transaction is then expanded to contain its readset. The transaction thus acquires only *Yellow* locks during lock acquisition, and as a result has an empty After set. This transaction is guaranteed to pass validation, avoiding the livelock problem. Also it never waits for any other transaction to complete lock inheritance. It may still, however participate in deadlocks. But note that to detect the deadlocks we are timing out and aborting a transaction that waits for transactions in its After set. Since the starvation prone transaction has an empty After set it will not get aborted even if it participates in a deadlock. Thus it is guaranteed to run to completion.

7 Discussion.

The *Five Color* protocol differs significantly from the 2-phase locking protocol in the way the precedence ($\rightarrow$) relation may grow. In the 2-phase locking protocol the precedence arcs are created when a transaction locks a data item. When a transaction locks an item (or upgrades a lock) it places itself after some other transaction, never before. Thus we say the precedence relation grows only in the *forward* direction. In fact this is the property of the 2-phase locking protocol that ensures serializability.

In our protocol, a path under the precedence order can grow in *both* directions. Suppose transaction T has reached its locked point and possesses a *Yellow* lock on x. A new transaction τ arrives and gets a *Green* lock on x. When τ reaches locked point, the arc $\tau \rightarrow T$ is created. Now, even after T terminates, as long as τ is active, τ' may come and place itself before τ, and hence before T. In this way, it is possible for future transactions to be logically placed in the past.[†]

Thus the basic mechanism by which 2-phase locking prevents serializability is not present in our protocol. Serializability is ensured, in this case by the *Blue* and *White* locks, and the validation procedure. Intuitively, if $T_1 \rightarrow \ldots \rightarrow T_2$ is a chain of transactions, then T_1 "knows" about $Rd(T_2)$ and $Wr(T_2)$, because it has *White* and *Blue* locks, respectively, on these data items. If any transaction τ attempts to read any data item in $Wr(T_2)$ (or write any item in $Rd(T_1)$) then due to the "triggering" caused by *Green* (*Yellow*) locking of a *Blue* (*White*) locked item, T_1 becomes a member of $Before(\tau)$ and T_1 inherits *White* (*Blue*) locks on $Rd(\tau)$ ($Wr(\tau)$). Thus information about the read and write sets flow up a chain in the form of "inherited" *White* and *Blue* locks. Now if τ may cause a cycle in the $\rightarrow$ relation by attempting to read an item *Yellow* locked by T, then T would become a member of $After(\tau)$ and violate the validation constraint. The other cases of information flow when the chain grows in the reverse direction, or when two chain as concatenated by a transaction is similar and is covered in the proofs (Lemma 4).

7.1 Efficiency Comparisons of Locking Protocols

[†] Setting *Green* locks on items which are *Yellow* locked may seem similar to a multiversion scheme with two versions [BeGo83]. The transaction with the *Yellow* lock has created a new version, but the transaction with a *Green* lock sees an older version. However we do not classify this protocol as a multiversion protocol as the same situation exists in 2-phase locking schemes, and other schemes where actual writing is done at commit time after locks are upgraded. The before/after value phenomenon is in general not classified as a true multiversion situation.

A method of comparing the performance of protocols is to compare the relative sizes of the set of logs they allow as output. If the set of logs generated by one protocol is a subset of the logs generated by another protocol, other, then the latter protocol is superior. However the set of output logs, produced by the *Five Color* protocol and the 2-phase locking protocols are incomparable. Intuitively our protocol should provide more concurrency for the following reasons (some of them have been stated earlier).

- Early release of read locks.

  The Five Color protocol releases read locks as soon as the data is read. This allows other transactions to update these data items without having to wait for the reading transaction to commit. (We assume for transparent 2-phase locking, all locks are held till commit point).

  Holding of read locks for a long time, as needed in the 2-phase locking protocol, has a not very obvious problem. Suppose T_1 and T_2 have read locks on x and T_3 requests an exclusive lock. After that T_4 requests a read lock. Can the request from T_4 be satisfied while T_3 is waiting? The answer is no. Because if T_4 is allowed to read x, then T_3 may get starved by read traffic on x, and never get the exclusive lock. Thus T_4 has to wait for T_1,T_2 and T_3 to commit (although it really could have read x when it asked for the lock.) This is an example where concurrency is drastically reduced by fairness constraints.

  The situation is different if read locks guaranteed to be held for a short time, as in the *Five Color* protocol. In this case we can allow T_4 to read before T_3 gets its *Yellow* lock as chances of starvation of T_3 is lower (as locks will clear fast). But if we choose to be fair, and make T_4 wait until T_3 gets the *Yellow*, T_4 can still get the read lock as soon as T_4 gets the *Yellow*. Also neither T_3 nor T_4 has to wait for T_1 and T_2 to commit, T_1 and T_2 will release the *Green* locks as soon as they complete reading x.

- Allowing reading of data items to be written later.

  This allows reading transactions faster access to data items that would have been exclusively locked for quite some time otherwise. Another advantage is typified in the above paragraph.

- Non-compatibility of *Yellow* locks.

  This property prevents some deadlocks. Consider variant of the 2-phase locking protocol that allow lock upgrading. This protocol allows reading of data items that will be updated later. But it also allows two transaction to begin updating the same data item, i.e. they may both read the data item and then request for upgrading to an exclusive lock. This causes what we term *trivial deadlock*. The deadlock is trivial to detect but just as serious as any deadlock,

as one of the transactions have to be aborted. The incompatibility of the *Yellow* lock, avoids chances of trivial deadlocks and causes delay of an update transaction while another is updating the data item (delay is better than deadlock). A simulation study we conducted showed that trivial deadlocks are the most common form of deadlock in the 2-phase locking protocol.

- Preordering of locking requests.

  This avoids deadlocks due to locking. The avoidance of such deadlocks as well as trivial deadlocks causes the *Five Color* protocol to have a lower chance of deadlocks. Thus we expect more throughput.

It seems that the *Five Color* protocol should perform better than the 2-phase locking protocols due to the above advantages. Here we present some characteristic situations that illustrate some situation where the *Five Color* protocol may has advantages over two variants of the 2-phase locking protocol. The examples consist of an input logs, and the corresponding outputs produced by three concurrency control strategies. The strategies are:

A: The standard 2-phase locking protocol.

B: A 2-phase locking variant, in which the read locks are acquired when the transaction reads, the transaction writes are kept in local storage, and write locks are acquired at commit time.

C: The *Five Color* Protocol.

input log 1	$W_1(x)$ $R_2(x)$ $W_1(y)$
A	$W_1(x)$ $W_1(y)$ $R_2(x)$.
B	$R_2(x)$ $W_1(x)$ $W_1(y)$.
C	$R_2(x)$ $W_1(x)$ $W_1(y)$.
input log 2	$R_1(x)$ $W_1(x)$ $R_2(x)$ $W_2(x)$ $W_1(y)$
A	$R_1(x)$ $W_1(x)$ $W_1(y)$ $R_2(x)$ $W_2(x)$
B	Deadlock
C	$R_1(x)$ $W_1(x)$ $W_1(y)$ $R_2(x)$ $W_2(x)$
input log 3	$W_1(y)$ $R_2(x)$ $W_2(x)$ $W_1(x)$ $R_2(y)$
A	Deadlock
B	Deadlock
C	$W_1(x)$ $W_1(y)$ $R_2(x)$ $R_2(y)$ $W_2(x)$

For input log 1, A) delays T_2 unnecessarily while B) and C) produce the same result. Input log 2 causes B) to enter a trivial deadlock. Both input logs are serializable.

Input log 3 is not serializable. Both A) and B) avoids illegal executions by causing deadlocks. However C) accepts the log, and rearranges the steps to provide a serializable output. The rearranging of the write steps are illustrative of the order of upgrading of *Yellow* locks to *Red* locks. T_2 is somewhat delayed, but that is more acceptable than a deadlock.

These are just a few examples to show cases where C) outperforms 2-phase locking. However it makes clear why comparisons get complicated and cannot be handled with the theoretical methods available to us at the moment.

7.2 Simulation Results

Until now the claims of higher concurrency and lower deadlock rates of the *Five Color* protocol were made based on intuitive reasoning. As a next step we present simulation results that compare the performance of the *Five Color* protocol against the 2-phase locking protocol and against the performance of a database which uses no concurrency control [Da84]. Using no concurrency control of course leads to incorrect executions, but it defines the upper bound on performance of concurrency control protocols.

Some of the results obtained in the simulations are described briefly in this section. We tested the protocols for throughput and abort rates at low to high loads (5 to 20 simultaneous transactions) and low and high conflict rates (by varying the read to write ratios and the number of data items in the database.)

As far as throughput is concerned, at low load and low conflict cases, the 2-phase locking protocol was about 6% poorer than no concurrency control, and the *Five Color* protocol was about 3% poorer than 2-phase locking.

At medium load and low conflict situations, the *Five Color* protocol was about 12% poorer than no concurrency control and the 2-phase locking protocol trailed the *Five Color* protocol by about 9%.

At medium load high conflict situations the *Five Color* was down by 23% from no concurrency control. The 2-phase locking protocol was plagued by thrashing due to trivial deadlocks and trailed the *Five Color* protocol between 10% to 20% for various test runs.

At high conflict and high load situations, the *Five Color* was worse than no concurrency control by 36% and the 2-phase locking protocol was lower than the *Five Color* protocol by about 35% to 40%.

The *Five Color* protocol did very well as far as avoiding deadlocks and having low abort rates. Without going into a case by case analysis, the *Five Color* Protocol had about one-fifth the number of aborts as 2-phase locking at low load and about one-tenth the number at high loads. The 2-phase locking protocol was plagued by a large number of trivial deadlocks and cyclic aborts at medium to high loads.

The only situation where the *Five Color* protocol was (marginally) poorer than the 2-phase locking protocol was when the load was quite low. This was due to the "bursty" way the *Five Color* protocol does I/O. For each transaction, all the data items are read at the onset and then all the data items are written at the end. While the transaction executes it consumes only CPU cycles. This makes the system oscillate between an I/O bound state and a CPU bound state. We noticed this phenomenon by checking the CPU and I/O queues. When there are several transactions running concurrently, the different times at which each transaction starts and terminates does even out the large I/O and CPU bursts, but at low loads there are

not enough transactions to give rise to a well balanced system. The 2-phase locking protocol on the other hand allows transactions to read data items interspersed with processing and thus has a better system performance at low loads.

Overall, the simulation results well support our claim of better performance and lower abort rates for the *Five Color* protocol.

7.3 Conclusions

We have presented a locking protocol that uses an unconventional locking strategy, and knowledge about the read and write sets of the transactions to allow non-2-phase locking on a general database. We show that this protocol ensures serializability and allows provides higher concurrency than the 2-phase locking protocol.

The *Five Color* protocol is substantially more complicated than the 2-phase locking protocol. In fact the simplicity and elegance of the 2-phase locking protocol is one of its major attractions. Nevertheless, we believe that the *Five Color* protocol is worth serious consideration.

The overhead involved in running the *Five Color* protocol comprises of somewhat increased CPU based processing. Since database applications are I/O intensive, database systems have significant I/O traffic and not enough CPU traffic. The extra overhead caused by the *Five Color* protocol algorithms is CPU based. All the lock tables and sets can be stored in memory, and the protocol could use the CPU cycles that would otherwise be idle due to I/O delays. The the extra overhead is hence not expected to cause reduction in system throughput.

Thus we conclude that it is possible for the *Five Color* Protocol to achieve more concurrency and by anticipating the *absence* of conflicts in a large number of cases, due to the information available to the transaction manager about the transaction readsets and writeset.

8 Bibliography

[BeGo80] Bernstein, P.A. and Goodman, N. *Fundamental Algorithms for Concurrency Control in Distributed Systems*, CCA Tech Report, CCA-80-05.

[BeGo82] Bernstein, P.A. and Goodman, N. *Concurrency Control Algorithms for Multiversion Database Systems*. Proc. ACM SIGACT/SIGOPS Symp. on Principles of Distributed Computing, (1982)

[BeGo83] Bernstein, P.A. and Goodman, N. *Multiversion Concurrency Control - Theory and Algorithms*. ACM Transactions on Database Systems, Dec. 1983.

[BeShRo80]Bernstein, P.A., Shipman, D.W., and Rothnie Jr., J.B. *Concurrency Control in a System for Distributed Databases (SDD-1)*. ACM Trans. on Database Systems **5**:1, pp. 18-51 (1980).

[BeShWo79]
Bernstein, P.A., Shipman, D.W., and Wong, W.S. *Formal Aspects of Serializability in Database Concurrency Control*. IEEE Trans. on Software Eng., BSE-5:*3, pp. 203-216 (1979).*

[Ca81] Casanova, M.A. *The Concurrency Control Problem of Database Systems*. Lecture Notes in Computer Science, Vol. 116, Springer-Verlag, 1981.

[Da83] Date, C.J. *An Introduction to Database Systems*, Vol 2., Addison-Wesley (1983).

[DaKe83] Dasgupta, P. and Kedem, Z. M. *A Non-2-Phase Locking Protocol for General Databases*. (extended abstract), 8th international Conf. on Very Large Data Bases, Oct 1983.

[Da84] Dasgupta, P. *Database Concurrency Control: Versatile Approaches to Improve Performance*. Ph.D. dissertation, State Univ. of New York, Stony Brook, Sept. 84.

[EsGrLoTr76]
Eswaran, K.E., Gray, J.N., Lorie R.A., and Traiger, I.L. *On notions of Consistency and Predicate Locks in a Database System*, Comm. ACM **14**:11, pp. 624-634 (1976).

[FiMi82] Fischer, M.J. and Michael, A. *Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network*. Proc. ACM SIGACT/SIGMOD Symposium on Principles of Database Systems, (1982).

[Ga83] Garcia-Molina, H. *Using Semantic Knowledge for Transaction Processing in a Distributed Database*. ACM Trans. on Database Systems, pp. 186-213, (1983).

[Gr78] Gray, J.N. *Notes on Database Operating Systems*. IBM Research Report RJ2188 (1978).

[GrLoPuTr76]
Gray J., Lorie R.A., Putzolu F., Traiger I. *Granularities of Locks and Degrees of Consistency in a Shared Database*. in *Modeling in Data Base Management Systems*, Nijssen, G.M. (ed.), North Holland Publishing (1976).

[KaPa81] Kanellakis, P.C. and Papadimitriou, C.H. *The Complexity of Distributed Concurrency Control*. Proc. **22**nd FOCS, pp. 185-197 (197).

[KeSi79] Kedem Z., Silberschatz A. *Controlling Concurrency Using Locking Protocols*. Proc. 20th IEEE FOCS, pp. 274-285 (1979).

[KeSi81] Kedem Z., Silberschatz A. *A Non-2-Phase locking Protocol with Shared and Exclusive Locks*. Proc. Conference on Very Large DataBases, Oct'81

[KeSi82] Kedem Z., Silberschatz A. *A Family of Locking Protocols Modeled by Directed Acyclic Graphs*. IEEE Trans. Software Engg. (1982) pp. 558-562.

[KuRo81] Kung, H.T.and Robinson, J.T. *On Optimistic Methods for Concurrency Control*. ACM Trans. on Database Systems **6**:2, pp. 213-226 (1981).

[Li79] Lindsay B.G. et al., *Notes on Distributed Database Systems*. IBM Research Report RJ2571(33471) 7/145/79.

[Pa79] Papadimitriou, C.H. *The Serializability of Concurrent Database Updates*, J. ACM **26**:4, pp. 631-653 (1979).

[PaKa82] Papadimitriou, C.H. and Kanllakis, P.C. *On Concurrency Control by Multiple Versions*. Proc. ACM Symp. Principles of Database Systems, March 1982.

[RoStLe78]Rosenkrantz, D.J., Stearns, R.I, and Lewis II, P.M. *System Level Concurrency Control for Distributed Database Systems*. ACM Transactions on Database Systems, **3**:2 pp. 178-198 (1978).

[SiKe80] Silberschatz, A. and Kedem Z.M. *Consistency in Hierarchical Database Systems*. J. ACM, **27**:1, pp. 72-80 (1980).

[Ul82] Ullman J.D. *Principles of Database Systems*., 2nd ed., Computer Science Press (1982), Potomac MD.

[Ya82] Yannakakis, M. *A Theory of Safe Locking Policies in Database Systems*, J. ACM **29**:3, pp. 718-740 (1982).

NYU COMPSCI TR-213
Dasgupta, Partha
The five color concurrency
control protocol c.2